Nicholas Chan
nipchan@ucsc.edu
5/09/2021
CSE13s Spring ‘21

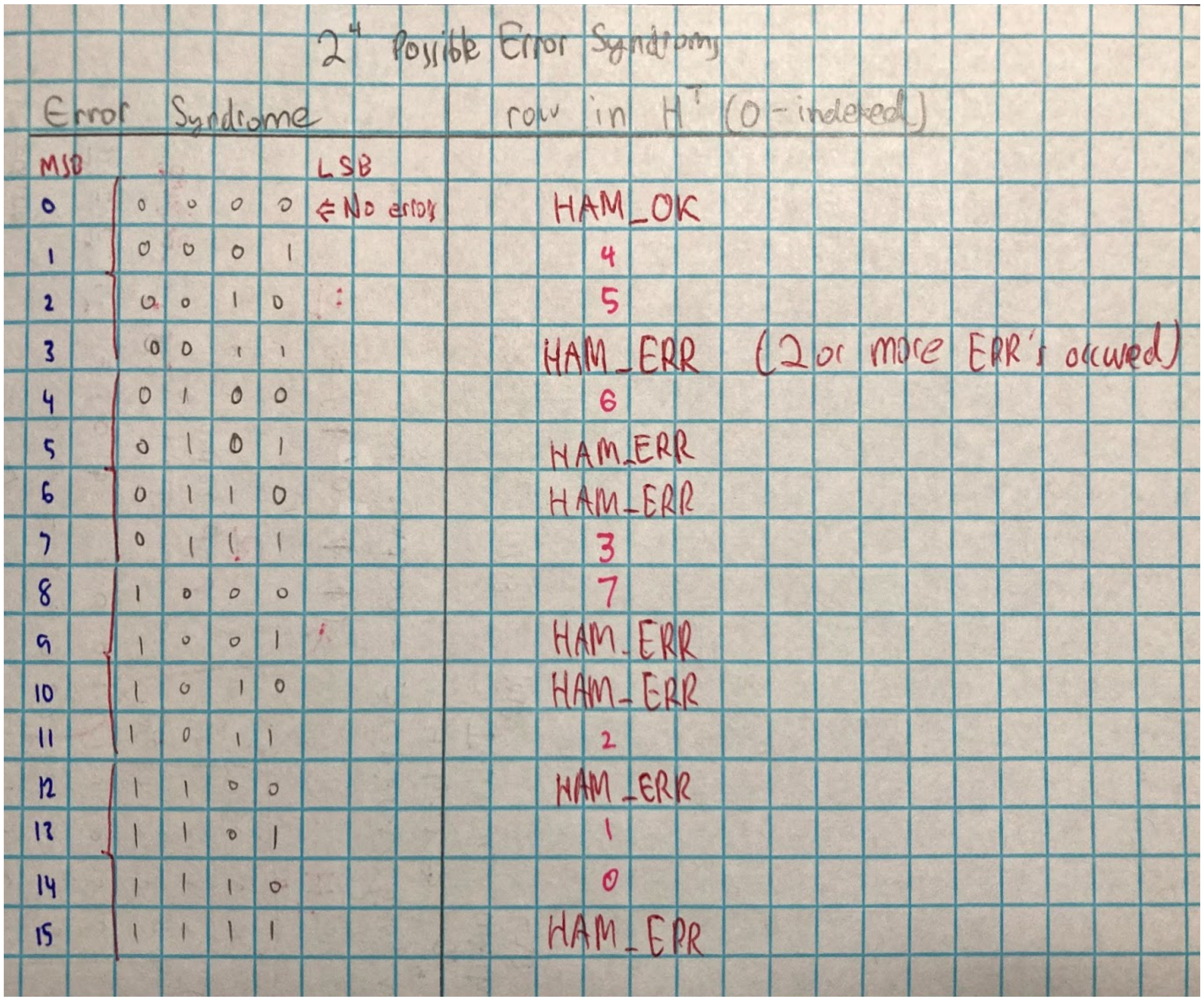
Design Document:
Assignment 5: Hamming Codes

Hamming codes are encoded messages that, depending on the format of hamming code, add a number of parity bits to a 4 bit unit known as a nibble. Nibbles are taken from splitting the bits of a character. As a result of this encoding process, hamming codes come in pairs, with one code holding the upper nibble of a message and another holding the lower nibble. The parity bits added to each nibble act as an error correction measure against bitflips caused by noise.

In assignment 5, I have been given the task of implementing an encoder and decoder for hamming code as well as ADTs for bit vectors and bit matrices. Encoding is a process which will require either the upper or lower nibble of a message character (1x4) to undergo matrix multiplication with a hamming code generator matrix (4x8). The product will be a hamming code (1x8), with the first 4 items corresponding to message bits and the last 4 items corresponding to parity bits. The decoding process will require the matrix multiplication of a hamming code (1x8) and the transpose of the generator matrix (8x4). The product of this will be the error syndrome of a hamming code (1x4). The error syndrome will be used to determine the HAM_STATUS of the hamming code.

Pre Lab Questions:

1.
    a. Below is the completed look-up table

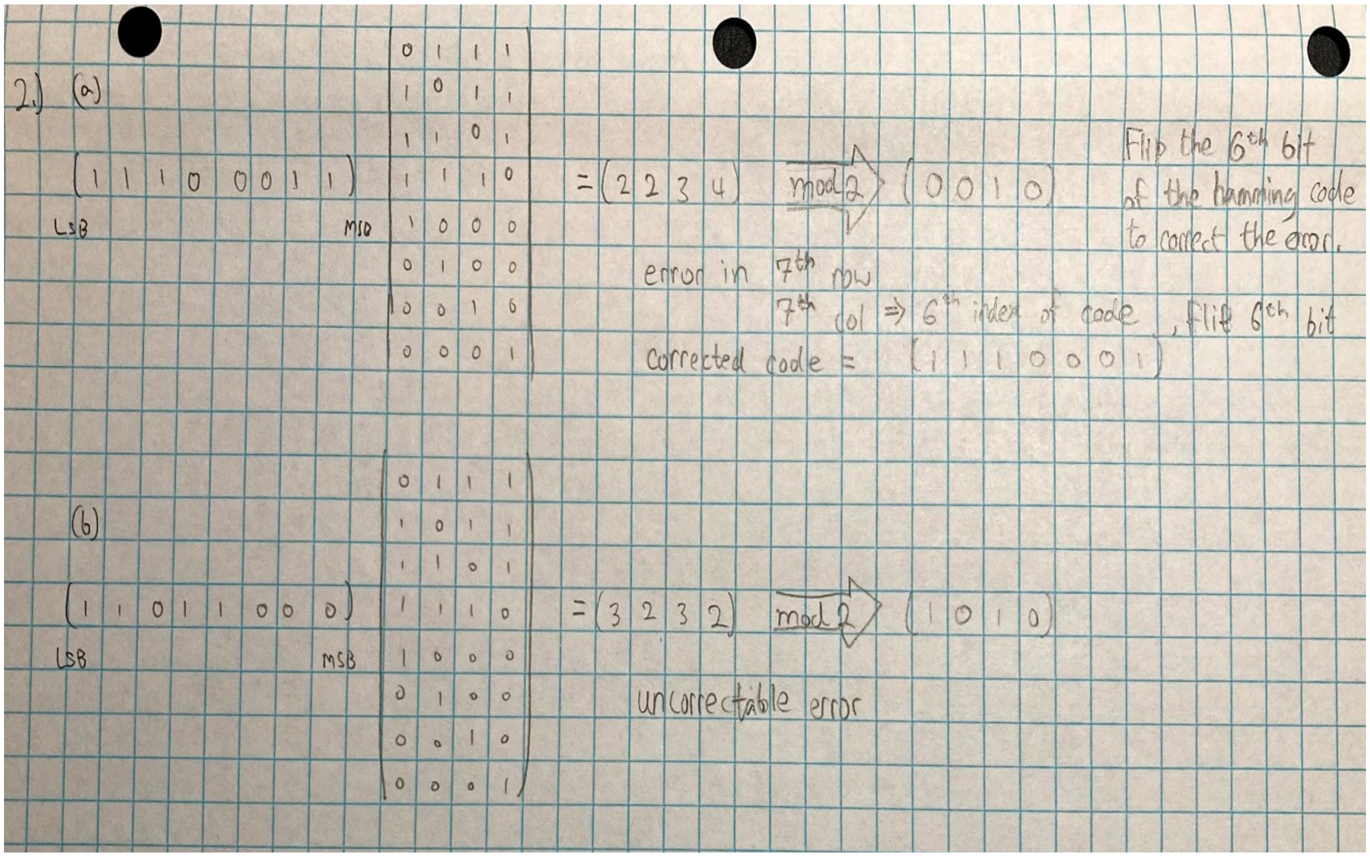


2^4 Possible Error Syndromes

Error Syndrome						row in H^T (0-indexed)
MSB				LSB		
0	0	0	0	0	⇐ No errors	HAM_OK
1	0	0	0	1		4
2	0	0	1	0		5
3	0	0	1	1		HAM_ERR (2 or more ERR's occured)
4	0	1	0	0		6
5	0	1	0	1		HAM_ERR
6	0	1	1	0		HAM_ERR
7	0	1	1	1		3
8	1	0	0	0		7
9	1	0	0	1		HAM_ERR
10	1	0	1	0		HAM_ERR
11	1	0	1	1		2
12	1	1	0	0		HAM_ERR
13	1	1	0	1		1
14	1	1	1	0		0
15	1	1	1	1		HAM_ERR

2.

a. For part a I found that the hamming code had an error that could be corrected by performing a bit flip on its 7th element from the left or, in code, its 6th index position.
b. For part b I found that the hamming code had an uncorrectable error. The error syndrome that I was left with after matrix multiplication corresponded to a HAM_ERR.



2.) (a)

$$\underset{\text{LSB} \qquad\qquad\qquad \text{MSB}}{\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 & 3 & 4 \end{pmatrix} \xrightarrow{\text{mod } 2} \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}$$

Flip the 6th bit of the hamming code to correct the error.

error in 7th row

7th col $\Rightarrow$ 6th index of code, Flip 6th bit

corrected code = $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$

(b)

$$\underset{\text{LSB} \qquad\qquad\qquad \text{MSB}}{\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 3 & 2 \end{pmatrix} \xrightarrow{\text{mod } 2} \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

uncorrectable error

Top Level:

Encoder

```
While fgets infile != EOF
    Read input(Characters)
    Split into
        message_nibble1, message_nibble2

    BitVector message_nibble
    BitMatrix generator matrix

    Mat_mul(message_nibble, generator matrix)

    write hamming_code1 (lower nibble)
    write hamming_code2 (upper nibble)
```

Decoder

```
Count = 0
While fgets infile != EOF
    Read input(hamming_code1, hamming_code2)

    BitMatrix transpose generator matrix

    Mat_mul(hamming_code1, transpose generator matrix)
         Get lower nibble (when count%2==0)
    Mat_mul(hamming_code2, transpose generator matrix)
         Get upper nibble (when count%2==1)

    Reconstructed byte = bitwise or (lower nibble, upper nibble) (when count%2==1)
    Write out reconstructed byte character (when count%2==1)
```

Hamming Module

```
Ham_encode(Matrix G, msg)
    Make bit matrix from msg nibble
    Matrix multiple msg nibble matrix by Generator matrix G to get Hamming code
    Turn Hamming Code into uint8_t  (4 data bits, 4 parity bits)
    Return Hamming code
(Ham_decode kinda functions like pop/peek for the stack ADT)

Ham_decode(Matrix Ht, Hamming Code, pointer to byte)
    Make Bit Matrix from Hamming Code
    Matrix Multiply Code by transpose of parity checker matrix
    Receive error syndrome Matrix and convert to uint8_t
    With data form of error syndrom, check HAM_STATUS cases
    If HAM_STATUS == 0
        Pointer to byte = uint8_t hamming code
        Return HAM_OK
    Else if lookup[HAM_STATUS] != HAM_ERR
        Pointer to byte = correct(uint8_t hamming code)
        Return HAM_CORRECT
    Else
        Pointer to byte = uint8_t hamming code
        Return HAM_ERROR
```

Design Process:
Coming into this class with the BME160 prereq, I realized I needed to acquaint binary, hexadecimal, little and big endian pretty fast, so some of my preparation for this lab ended up being a study day for those things. I also watched Eugene's and Sahiti's sections to get a better grasp of how bit vectors were formed and manipulated. One of the first things I did before actually touching code for this assignment was to try a "Hamming encode" and "decode" by hand through the matrix multiplication formula layed out for the Hamming (8,4) scheme. The Hamming (8,4) scheme served as the basis to this assignment and was an encoding pattern that allocated a nibble for data bits and a nibble for parity bits. The next thing I did before coding was the construction of a lookup table for the Hamming (8,4) scheme. The lookup table would be used to speed the process of correcting erroneous bit flips or returning HAM_STATUSES (HAM_OK. HAM_CORRECT, or HAM_ERR which correspond with good code, correctable code, code with >1 error thus making it uncorrectable code).

1. The first ADT I constructed was the BitVector
    a. This bv struct consisted of a byte array (using uint8_t items) and a uint32_t value for length
    b. Each byte item would serve as a bit array with a length of 8 bits

2. The main functions for facilitating Hamming encode and decode were the functions for setting, clearing and getting a bit.
    a. Each of these functions worked by finding a byte in the byte array to manipulate and then performing a bitwise operation on the desired bit array (byte)
        i. Set bit used bitwise or, bit clear used a bitwise and of a complement and get bit used a bitwise and operation.
        ii. The bit vector XOR function was particularly useful as it was the equivalent of addition with a modulus 2 condition, which the matrix multiplication in the encoding and decoding processes used
            1. That being said, bitwise and was used as multiplication with a modulus 2 condition in the same context as XOR.
3. The next ADT I constructed was the BitMatrix
        i. This bv struct consisted of a bit vector ADT which act as an actual matrix throughout the bv.c module, a uint32_t for rows and a uint32_t for columns.
        ii. Each byte item in the bm bit vector would serve as a bit array with a length of 8 bits
        iii. BitMatrix functions for setting, clearing and getting a bit relied on their BitVector counterparts.
        iv. A function named bm_from_data would create a BitMatrix given a uint8_t
            1. This was done by setting bits in a bit matrix item for each bit that existed in the starting uint8_t
            2. This would be repeated for all the items in the bit matrix
        v. A function named bm_to_data would create a uint8_t given a BitMatrix
            1. This was done by setting bits in a BitMatrix byte item for each position in the uint8_t that held a bit
        vi. Bm_multiply was a function that takes two BitMatrix instances and performs matrix multiplication with a modulus 2 condition
            1. This required a for loop over the rows of the product matrix, a for loop over the columns of the product matrix and a for loop over the common factor between the two starting matrices.
4. I then moved onto constructing the hamming.c module which held ham_encode and ham_encode
    a. ham _encode functioned by facilitating a matrix multiplication between a message nibble in the form of a uint8_t message (which would be converted to a bit matrix) and the generator matrix to produce a hamming code
    b. ham _decode functioned by facilitating a matrix multiplication between a hamming code (which would be converted to a bit matrix) and the transpose of the parity checker matrix to produce an error syndrome. The error syndrome

produced would determine the next course of action (continuing or correcting an erroneous bitflip)

5. Encode and Decode functioned in tandem, with encode receiving data a byte at a time, encoding each byte into an upper and lower nibble, converting those nibbles into hamming code and finally sending the hamming codes to output. Encode was implemented as a usual command line argument program that relied on a while loop with fgetc != EOF as the main condition. Decode would receive data by the byte in a similar fashion to encode.c. However, decode was required to take in hamming codes with the lower and upper nibble codes alternating. This required a condition that relied on modulus 2 to determine the correct time to pack bytes and send characters to output.

I thought this assignment was pretty challenging especially with the need for memory management and working with bits. I also felt that I ran into some pitfalls with fgetc as I had an incident where fgetc would == EOF everytime the character ‘o’ appeared in the file stream. This took some time to understand and was a quick fix that ended up costing me a lot of time. I really appreciated the code given to use which exhibited clean implementation of command line arguments for input and output. I haven’t seen too many clean implementations yet as I have just been going off of what I understood.

References:

- https://en.wikipedia.org/wiki/Bit_array
- http://sticksandstones.kstrom.com/appen.html